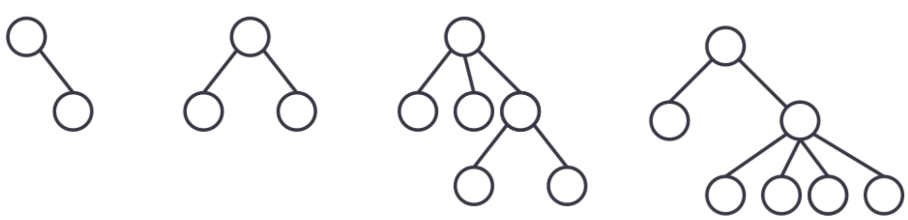
1. Calcolare la complessità $T(n)$ del seguente algoritmo MYSTERY assumendo implementazione della struttura UnionFind tramite quickFind:

Algorithm 1: MYSTERY(INT n) $\rightarrow$ INT

```
UNIONFIND uf ← new UNIONFIND()
for j ← 1 to n do
    uf.makeSet(i)
end
INT u ← 1
INT v ← n
while u ≤ n do
    uf.union(uf.find(u), uf.find(v))
    u ← u × 2
    v ← v − 1
end
return uf.find(1)
```

Soluzione Innanzitutto, si deve tenere in considerazione il fatto che secondo l'implementazione quickFind, le operazioni sulle strutture UnionFind hanno la seguente complessità: *makeSet* e $find$ hanno costo costante $O(1)$, mentre *union* ha costo $O(n)$ nel caso pessimo (con n dimensione della struttura). Veniamo ora all'analisi della complessità $T(n)$ dell'algoritmo. L'algoritmo esegue operazioni di complessità costante ad esclusione dell'operazione *union*. Il primo ciclo esegue una quantità pari ad n di operazioni a costo costante, quindi con complessità risultante $O(n)$, e costruisce una struttura UnionFind di dimensione n. Il secondo ciclo viene eseguito una quantità di volte pari a $\lceil \log n \rceil$. Il corpo del ciclo, nel caso pessimo, ha costo $O(n)$ visto che oltre ad operazioni di costo costante include l'operazione *union* eseguita sulla struttura UnionFind di dimensione n costruita dal primo ciclo. Complessivamente, tale secondo ciclo ha quindi costo $O(n \log n)$. Avremo quindi $T(n) = O(n) + O(n \log n) = O(n \log n)$.

2. Si scriva un algoritmo che prende in input un albero n-ario T e conta quanti sono i livelli che hanno un numero di nodi pari. Nei casi seguenti quindi restituisce rispettivamente 0, 1, 1, 2.



Soluzione L'esercizio si risolve con una visita in ampiezza. Nella coda usata per visitare i nodi si memorizza, oltre al valore del nodo, anche il livello in cui si trova. Ogni volta che si estrae un nodo si verifica se si è raggiunto un nuovo livello. In questo caso si verifica se i nodi del livello precedente erano in numero pari e il conteggio riparte per il nuovo livello. In caso contrario si aumenta il contatore dei nodi al livello corrente. Necessario un controllo finale per verificare se l'ultimo livello ha un numero di nodi pari.

Chiamiamo tale algoritmo CONTALIVELLIPARI. Visto che tale algoritmo richiede la visita dell'intero albero, e che per ogni nodo si eseguono operazioni di costo costante, il costo computazionale risulta essere $\Theta(n)$, con n dimensione dell'albero.

Algorithm 2: ContaLivelliPari(Nodo T) → Int

```
Int livelliConNodiPari ← 0
Queue q ← new Queue()
q.enqueue([T, 0])
Int livelloCorrente ← 0
Int nodiLivelloCorrente ← 0
while q.first ≠ null do
    /* estrae dalla coda una nuova coppia [N, l]                              */
    [N, l] ← q.dequeue()
    if l ≠ livelloCorrente then
        /* N è il primo nodo di un nuovo livello l                            */
        if nodiLivelloCorrente % 2 = 0 then
            livelliConNodiPari ← livelliConNodiPari + 1
        end
        nodiLivelloCorrente ← 1
        livelloCorrente ← l
    else
        /* N è un ulteriore nodo dell'attuale livello corrente                */
        nodiLivelloCorrente ← nodiLivelloCorrente + 1
    end
    for x ∈ N.children do
        q.enqueue([x, l + 1])
    end
end
/* controllo relativo all'ultimo livello                                      */
if nodiLivelloCorrente % 2 = 0 then
    livelliConNodiPari ← livelliConNodiPari + 1
end
return livelliConNodiPari
```

3. Progettare un algoritmo che dati due vettori di numeri $A[1..n]$ e $B[1..n]$ calcola quanti indici $i \in \{1, \ldots, n\}$ sono tali che $A[i]$ appare anche in B.

Soluzione Una possibile soluzione prevede di ordinare il secondo vettore al fine di poter effettuare ricerche di elementi con costo logaritmico tramite ricerca binaria. Una volta ordinata sarà sufficiente scorrere gli elementi del primo vettore, e incrementare un contatore ogni volta che si incontra un elemento che è presente anche nel secondo vettore (usando appunto una ricerca binaria). L'algoritmo Conta utilizza la variabile ausiliaria *conta* come contatore. Inoltre, l'algoritmo usa un algoritmo di ordinamento Sort che non specifichiamo; assumiamo che sia un algoritmo di ordinamento ottimale (ad esempio heapsort) di complessità $O(n \log n)$.

Studiamo ora la complessità $T(n)$ dell'algoritmo Conta iniziando l'analisi dall'algoritmo ausiliario di ricerca binaria Ricerca. Tale algoritmo ha un costo $O(\log n)$. L'algoritmo Conta esegue prima l'ordinamento di costo $O(n \log n)$. Successivamente esegue un ciclo in cui invoca per n volte Ricerca, per un costo complessivo $O(n \log n)$. Complessivamente avremo $T(n) = 2 \times O(n \log n) = O(n \log n)$.

Algorithm 3: CONTA(INT $A[1..n]$, INT $B[1..n]$) → INT

```
Sort(B)
Int conta ← 0
for i ← 1 to n do
    if Ricerca(A[i], B, 1, n) then
        conta ← conta + 1
    end
end
return conta

/* funzione di ricerca binaria in array ordinato */
function Ricerca(Int x, Int V[1...n], Int s, Int e)
if s > e then
    return false
else
    Int m ← (s+e)/2
    if x = V[m] then
        return true
    else
        if x < V[m] then
            return Ricerca(x, B, s, m − 1)
        else
            return Ricerca(x, B, m + 1, e)
        end
    end
end
```

4. Si consideri un impianto di irrigazione che collega delle piante a vari rubinetti che possono erogare acqua. Tutti i rubinetti sono inizialmente chiusi, e bisogna capire quale rubinetto aprire per far arrivare più velocemente possibile l'acqua ad una data pianta che necessita di essere annaffiata. L'impianto è rappresentato tramite un grafo non orientato pesato $G = (V, E, w)$ in cui i vertici in V rappresentano rubinetti o piante, un arco $(u, v) \in E$ rappresenta un tubo di collegamento dal vertice u al vertice v, ed il peso $w(u, v)$ indica il tempo che l'acqua impiega per attraversare il tubo (u, v) (sotto l'assunzione che l'acqua impiega il medesimo tempo ad attraversare il tubo partendo dal vertice u o partendo dal vertice v). Progettare un algoritmo che dato il grafo non orientato pesato $G = (V, E, w)$, l'insieme $R \subseteq V$ dei rubinetti, e la pianta $p \in V$ da annaffiare, restituisce il rubinetto $r \in R$ da aprire per far arrivare il più velocemente possibile l'acqua alla pianta p.

   Soluzione Il problema prevede di trovare il vertice appartenente ad R che ha il cammino minore per raggiungere il vertice p. Essendo il grafo non orientato, questo coincide con il vertice appartenente ad R a distanza minima da p. I pesi saranno non negativi in quanto quantificano degli intervalli di tempo, quindi è possibile utilizzare l'algoritmo di Dijkstra.

   L'algoritmo ANNAFFIA è una versione dell'algoritmo di Dijkstra che visita i nodi in ordine di distanza crescente da p, e che interrompe l'esecuzione appena si raggiunge un nodo appartenente ad R. Se si termina l'esecuzione dell'algoritmo di Dijkstra senza raggiungere nodi appartenenti ad R, allora non è possibile annaffiare la pianta p e si restituisce un errore. In questo modo, la complessità dell'algoritmo ANNAFFIA nel caso pessimo coincide con la complessità dell'algoritmo di Dijkstra, ovvero $O(m \times \log n)$ dove $m = |E|$ e $n = |V|$.

Algorithm 4: ANNAFFIA(GRAFO $G = (V, E, w)$, SET[VERTEX] R, VERTEX p) $\rightarrow$ VERTEX

```
/* inizializzazione strutture dati */
n ← G.numNodi()
DOUBLE D[1..n]
for i ← 1 to n do
    D[i] ← ∞
end
D[p] ← 0
MINPRIORITYQUEUE[INT, DOUBLE] Q ← new MINPRIORITYQUEUE[INT, DOUBLE]()
Q.insert(p, D[p])

/* esecuzione algoritmo di Dijkstra */
while not Q.isEmpty() do
    u ← Q.findMin()
    if u ∈ R then
        return u
    end
    Q.deleteMin()
    for v ∈ u.adjacent() do
        if D[v] = ∞ then
            /* prima volta che si incontra v */
            D[v] ← D[u] + w(u, v)
            Q.insert(v, D[v])
        else if D[u] + w(u, v) < D[v] then
            /* scoperta di un cammino migliore per raggiungere v */
            Q.decreaseKey(v, D[v] − D[u] − w(u, v))
            D[v] = D[u] + w(u, v)
        end
    end
end
return error
```

The End.